SpaceOps-2025, ID # 55

Toward an Energy-Aware Ground Segment

José da Silva[a], Otto Kaaij[b], Luís Cruz[c]
[a] *Solenix Engineering GmbH , Germany,* *jose.silva@solenix.ch*
[b] *Delft University of Technology, Netherlands,* *otto.kaaij@gmail.com*
[c] *Delft University of Technology, Netherlands,* *L.Cruz@tudelft.nl*

Abstract

Energy consumption and energy-related requirements are seldom considered when discussing software development. This is the case also of spacecraft operations where software plays a critical role. Modern platform architectures have made significant steps in enabling the consumption, processing, curation, archival, and retrieval of vast amounts of data, and are continually being pushed to accommodate more requirements and ideas to provide a limitless foundation for AI. However, despite the advancements in platform and application development, energy-aware requirements are often overlooked. Unless energy becomes a constraint, software architects tend to neglect energy-related considerations when building their solutions, leading to a situation where hardware is simply scaled up until it runs satisfactorily. In this paper, we bring energy considerations to the forefront of ground segment software development, exploring a methodology called SMURF to measure the energy consumption of complex software tools. We also present a detailed experiment in which we study the energy consumption of a distributed parameter archive – MUST, used for housekeeping telemetry monitoring and offline analysis. The changes proposed to the system because of this study are carefully described, providing valuable insights for the future development of more energy-aware software solutions.

Keywords: Software; Green; Sustainability; Spacecraft Operations; Ground Segment Engineering;

Acronyms/Abbreviations

API	=	Application Programming Interface
EO	=	Earth Observation
ESA	=	European Space Agency
ESOC	=	European Space Operations Centre
FCT	=	Flight Control Team
HKTM	=	Housekeeping Telemetry
ICT	=	Information and Communication Technology
MCS	=	Mission Control System
MPS	=	Mission Planning System
MUST	=	Mission Utility and Support Tools
MUSTLink	=	MUST REST API
REST	=	Representational State Transfer
SMURF	=	Simulate and Measure to Understand Resource Footprints
TC	=	Telecommand
VRA	=	Viewport Resolution Aggregation

1. Introduction

Cloud architectures and services are often seen as a way to make software more sustainable, as datacentres have economy of scale incentives to run efficient hardware and not waste energy on idle and spare processes. Nevertheless, they also have the incentive to purchase cheap energy, which is often not renewable. Furthermore, the energy required to transport data between a datacentre located far away from the operations centre or even in a multi-cloud resilient environment is also overlooked. Energy-awareness shall not be simply delegated to external layers. Instead, it shall be embraced during all phases of software development leading to new opportunities for optimization. For example, by re-examining our assumptions about system design, we can ask ourselves whether our API servers need to be always-

on, or if we can optimize data processing and storage to reduce energy consumption. We can also explore the trade-offs between performance and energy efficiency, such as determining the minimal resources required to achieve a substantial gain in performance for long-term requests. By considering these factors, we can develop more energy-efficient software solutions that minimize their environmental impact.

Energy awareness in software development is crucial. As demand for computing power grows, so does the energy consumption of software applications, contributing to increased carbon footprints and environmental degradation. By prioritizing energy-efficient practices in software development, engineers can create applications align with global efforts to combat climate change but also enhance the longevity and scalability of the software systems, ultimately leading to a more sustainable digital future.

In this paper we present the results of the application a methodology called SMURF to measure the energy consumption of the various functions of a software application called MUST – a parameter archive used in the ground-segment of spacecraft operations.

1.1 Energy Consumption in Computing

It is no secret that computers consume energy. Users of personal laptops, tablets, mobile and wearable devices are particularly mindful of this fact, as they depend on batteries that deplete if not charged. In contrast, desktop users might be less mindful about power usage and energy drainage as these devices are permanently plugged into the wall. Today, computer usage is increasingly centred around the internet, where we continuously access systems deployed in datacentres located around the world, each powered by a different energy grid and mix.

While it may seem that hardware is solely responsible for energy consumption as electricity flows through wires and transistors, this perspective overlooks the critical role of software. The energy consumption of a computer system is not static: it depends on the load imposed by the software. “Heavier” software processes increase CPU load, raising temperatures and leading to higher cooling efforts. Additionally, operations such as writing to disk, accessing memory and network usage have direct impact on the hardware usage and, consequently, energy consumption.

1.2 The environmental impact of Information and Communication Technology (ICT)

In the midst of the climate crisis, we should not allow ourselves to forget that the ICT industry, the sum of all phones, laptops, datacentres, networking equipment, and all other computing devices, accounts for around 2.1% to 3.9% of global energy consumption [4]. That, as a part of this industry, we generate more than 50 million tons of e-waste per year when we throw away old hardware [5], and that we ruin entire ecosystems by mining for the rare metals needed to produce new hardware. And that we generally remain, wilfully or not, ignorant of this.

1.3 Why energy is not often a key driver in software design

While hardware manufacturers have made significant strides in developing more energy-efficient components, the question remains: Why aren’t software engineers more considerate of energy consumption in their designs?

Software design is often constrained by multiple dimensions traditionally specified as software requirements (or implicit in user-stories). For instance, software development is often influenced by its functional objectives, implementation timelines, development effort, infrastructure limitations, security, performance, usability and maintainability considerations. These dimensions are carefully considered, implemented and validated as part of our software products as they are key elements to deem the software as acceptable by its stakeholders. For instance, when software is used in contexts where battery drainage is not an issue then energy consumption is not a major concern. This is the case for spacecraft operations. Energy costs remain relatively low compared to the cost of software development labour for most cases, which influences software design to prioritize maintainability, configurability, scalability, and extensibility over energy efficiency.

1.4 Why some software is already energy-aware

Energy efficiency in software is commonly considered from design when it results from economical or technical drivers. For instance, developers of software for mobile and wearable devices are carefully measuring energy consumption [1] as battery drainage is a key economical-factor: e.g. users are less likely to install and use applications or run websites that quickly drain their device battery life [3]. In mobile and wearable applications, energy efficiency becomes a key constraint.

Thanks to the scale incentives, cloud providers are also engineering more energy-efficient cloud architectures. For example, serverless computing [2] allows resources to be scaled according to demand, promoting more efficient use of datacentre resources. However, these gains can sometimes be offset by the rebound effect, where increased efficiency leads to higher overall consumption due to overexploitation of spare capacity. Nonetheless, there is a direct correlation between infrastructure expenses and energy consumption: acquiring and maintaining a larger infrastructure has higher costs. Additionally, larger infrastructures require more energy to run effectively. These expenses are evident in the billing models of cloud services, where costs for CPU usage, memory, network, and storage are determined by expected and actual demand. Taking the infrastructure costs into account while designing and developing software expectedly translates into savings. For instance, software engineers can design their web services to be lazy loaded and instantiated only when there are requests, stopping the services completely during idle times. This could reduce the infrastructure bill while sparing energy waste.

Resource scarcity, climate change and power-grid related challenges are good enough reasons to introduce energy considerations as part of the software development processes, independently of the economical perspective. On the other hand, succeeding in aligning economical perspective with energy efficiency appears to be the key to unleash energy efficiency gains in software development. This is the case for both mobile software and large-scale infrastructure applications.

1.5 Demanding Energy Efficient Software

Expressing energy-related requirements or user stories can be challenging. One of the primary difficulties lies in quantifying the energy consumption of software. The energy profile of the same software can vary significantly depending on the hardware and operating system architecture it runs on. Moreover, even when executed on the same platform, the energy profile is not constant between different runs due to factors such as varying hardware temperatures, which directly impact energy usage. The complexity increases further: a specific algorithm may consume more energy on one CPU architecture (A) than on another (B), while a different algorithm might have the opposite behaviour.

These difficulties make it hard to express clear and objective requirements such as “Turn-by-turn navigation shall use 0.5 Kilowatt Hour (kWh)”. Instead, energy-related requirements are typically expressed as desires: “turn-by-turn guided navigation should not drain more battery than a car can charge.” [6].

1.6 Characterization of Software in Spacecraft Operations

Spacecraft operations are heavily dependent on computer software systems. Mission control and planning systems, simulators, operations automation and other auxiliary systems are commonly found as part of the ground segment suits from organizations executing spacecraft operations.

Spacecraft engineers and operators, during working hours are typically using their terminals plugged into the wall. Mobile and wearable applications are not prominent in this field. Performance, security and system reliability are among the most valued requirements for this type of applications. AI is increasingly more prominent in the control rooms: automated operations, anomaly detection and intelligent planning are footprints of this increase. Typically, several mission products are archived throughout the entire mission lifetime and even beyond with long term data preservation initiatives, contributing to a growing infrastructure.

Several organizations, including ESA play a major role in measuring and combatting climate change through its meteorological and climate observation missions. In this sense, extending this critical role to the ground segment itself is a direction worth exploring.

Empirically, we could hypothesize that software engineering for the ground segment does not strongly adhere to energy-awareness as most of the requirements available in public tenders and well-established systems do not include this dimension. Therefore, we engaged in an experiment, using a software tool – MUST (section 2.1). MUST was developed and maintained by the Operations Centre of the European Space Agency in collaboration with Solenix[*]- an independent, privately owned company focused on value-driven projects in the space domain including activities with the European Space Agency, EUMETSAT, national space agencies and private space companies.

Energy-driven requirements are not commonly present in the procurements of ground segment systems. The software composing the ground segment is complex, in some cases legacy, and the infrastructure hosting these systems is not commonly centralized, reducing the economic incentives for energy gains. For these reasons, spacecraft

[*] https://www.solenix.ch

operations presents as an environment where analysing energy hotspots and reflecting on potential solutions is likely to have significant impact with tangible contribution to carbon-efficiency.

2. Material and methods

In this paper, we are studying one application used by most of the major European spacecraft missions and programmes from the energy perspective. As such we rely on two main elements: the application itself (MUST) and the adopted methodology to study its energy profile - SMURF. These are introduced within this section.

2.1 Mission Utility and Support Tools (MUST)

MUST is a parameter archive with significant prominence at ESA/ESOC. It is actively used by over 250 engineers from several ESA missions and programmes [7], primarily at ESOC, the European Space Operations Centre but also from other establishments.

MUST consists of an ecosystem of components including the importation module, the archive, the data provision layer, a REST API, a bundle of web applications and many other features and utilities. MUST is not a monolith and most of its components are extensible and adaptable – parts of MUST are compatible with other applications that are equally prominent in the ground segment software. At ESA, the *Multimission-as-a-Service* initiative is based on a system called ARES [12], which database technology and design is very different from the database used by MUST. In any case, the upper layers of MUST (including the REST-API and web clients) are equally compatible with ARES for further reusability and collaboration.

MUST, stores parameters in calibrated form in the relational database that lies at its core. In addition to efficient data storage, MUST provide data analysis and flexible data visualization tools. Users can plot data, display it in table views, or configure custom alphanumeric displays. A synthetic parameter engine allows users to create new parameters on-the-fly by coding functions inline using JavaScript. In addition, users can use two data analysis tools, DrMUST [13] and Novelty Detection [14], to identify correlations and detect anomalies in telemetry data. These encompass the core functionalities of MUST: import, store, disseminate and visualize data.

Figure 1. Screenshot of NGWebMUST displaying a dashboard combining housekeeping telemetry displayed as single linear plots, textual data as Bar Charts, a XY Correlation plot and a table containing the history of events registered on-board.

MUST can import data from multiple sources and formats, utilizing various importers. The platform supports the importation of diverse data types such as housekeeping telemetry, event logs, telecommands, flight dynamics data, planning files and other data. These importers have been developed incrementally over time to meet the demands of numerous European space missions and programmes. Imported data is accessible via an API, called 'MUSTLink' through which the frontend called NGWebMUST – accesses data. The API can also be used to get data from the system programmatically. Over time, MUST has evolved into a complex system with many different uses and components. This is exacerbated by a mostly on-premise and decentralized deployment approach in which each mission hosts MUST under a different server and baseline setup.

2.2 SMURF

SMURF is a five-step methodology designed to assess the energy profile of a single, complex, multi-faceted software system using real usage data [8]. One of the primary advantages of SMURF over traditional energy-regression techniques is its incorporation of real-world application usage patterns. For example, if a particular functionality consumes a substantial amount of energy but is rarely utilized, addressing its energy consumption may yield minimal benefits for the overall application. In contrast, optimizing a frequently used functionality for energy efficiency, even though marginal improvements, could result in significant gains in overall energy performance. This perspective allows for a better understanding of cost-benefit analysis when addressing energy waste hotspots. In addition, as this study represents one of the first investigations into energy efficiency within ground segment software, the generation of impactful results is particularly valuable. Not only do these improvements translate into immediate and substantial energy savings, but they are also more easily communicated and perceived. This increased visibility can enhance interest in the topic and contribute to greater awareness of energy efficiency in software systems.

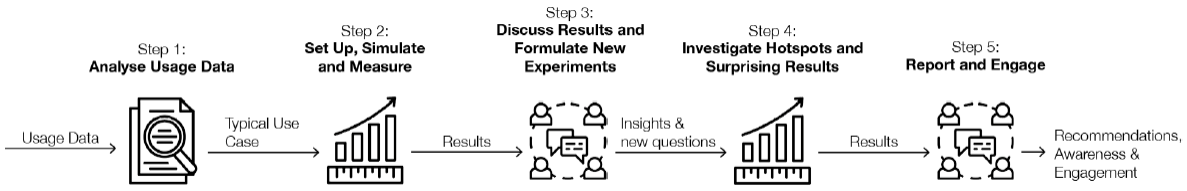

Figure 2. A graphical representation of the SMURF methodology [8]

The SMURF methodology has the following five steps:

- **Step 1 – Analyse Usage Data**: Take stock of the available usage data, and using it, formulate a typical use case over a representative period, that describes how much each component of the system is used, under typical use, in that period. Usage data is typically available under the form of application logs or software profiling and instrumentation databases.
- **Step 2 – Set up, simulate, and measure:** Set up a test deployment of the system, and use it to simulate the typical use case formulated in step one. Measure the energy consumption of the system throughout the simulation. This requires a test deployment of the system in a controlled environment that is measurable. For this we follow Cruz [9] and Mancebo, García, and Calero [10] for best practices on energy measurements. We need a system that runs Energibridge [11], the software under test (MUST in this case) and as little as possible otherwise.
- **Step 3 – Discuss results and formulate new experiments**: Evaluate the results, closely involving different stakeholders. Use their knowledge to gain an understanding of what results are surprising, and what aspects of the system could benefit from a deeper investigation. Formulate new experiments to answer those questions.
- **Step 4 – Investigate hotspots and surprising results:** Perform the experiments formulated in step 3.
- **Step 5 – Report and engage:** Report the results to various stakeholders. Use their insights to deepen the context around the results, and use the results to engage all kinds of different stakeholders with sustainable software ideas.

3. Results – Applying SMURF to MUST

We selected an Earth Observation (EO) mission as a case study for the energy profiling analysis of MUST. There are two arguments sustaining this option: There is a higher prominence of EO missions among the existing MUST installations and the operational dynamics of EO missions is actually perceived as demanding from the MUST

utilization point-of-view (more frequent passes, higher volumes of down and uplinked data, more users in parallel). That perception results from the empiric experience from MUST engineers who also have a higher volume of maintenance requests from these type of missions.

3.1 Step 1: Analyse Usage Data

In order to capture a month of nominal operations we used the following artefacts as input:

- **A log file containing a list of all the REST-API calls** executed by the users over the period of 1 month. Whenever a user executes any action in the application (e.g. clicking a button on the web client or calls the REST-API directly), the exact request is timed and logged in this file. This provides an exact understanding of what are the application users doing with it. What API calls are more often utilized and what are the arguments used in these requests.
- **Scheduler information**. The ingestion of telemetry data in MUST is triggered via scheduled tasks. For instance, every X minutes, the importer component will start and verify if new telemetry is incoming, ingesting it, if so.
- **One day of TM, TC and Event data**. We use this data to populate our simulated MUST environment to ensure we can orchestrate realistic responses when requests are placed to the system. We replicated this 1-day of data over a period of one week to ensure a representative archive.

Our typical use-case is based on a week of data collected from August 12th to August 18th, 2024. We chose this timeframe because the volume and nature of requests can vary significantly from day to day—Sunday, for instance, is typically much quieter than the following Monday—yet they remain relatively consistent from week to week. Figure 3 illustrates the number of requests per hour over a four-week period, from August 5th to September 1st. This figure reveals a distinct weekly pattern, characterized by fewer requests during nighttime hours and quieter days on weekends.

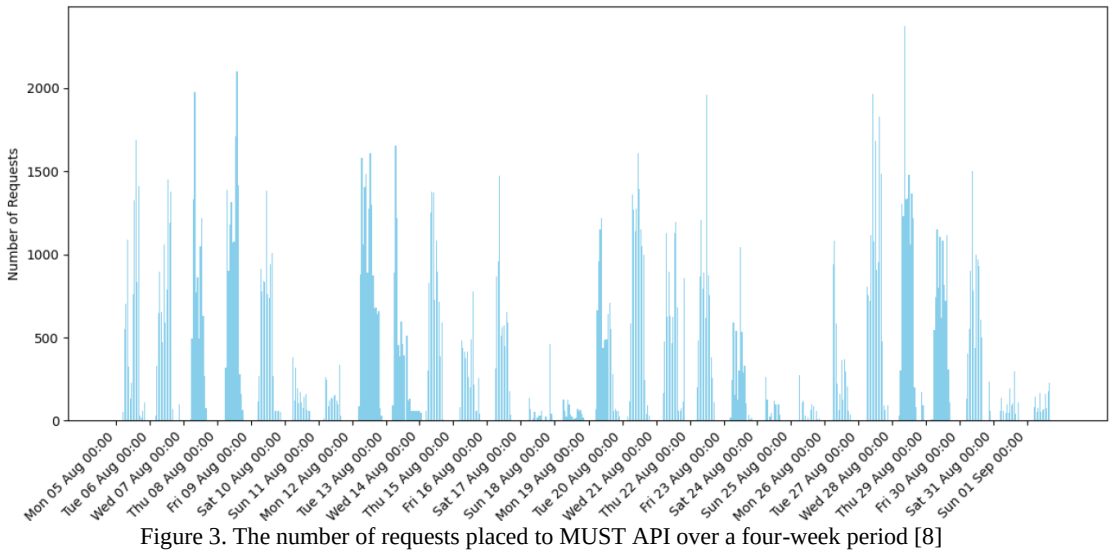

Figure 3. The number of requests placed to MUST API over a four-week period [8]

After grouping the requests from the selected week per type of API request it is possible to determine what is the effective weekly usage of each application feature (how many requests per feature).

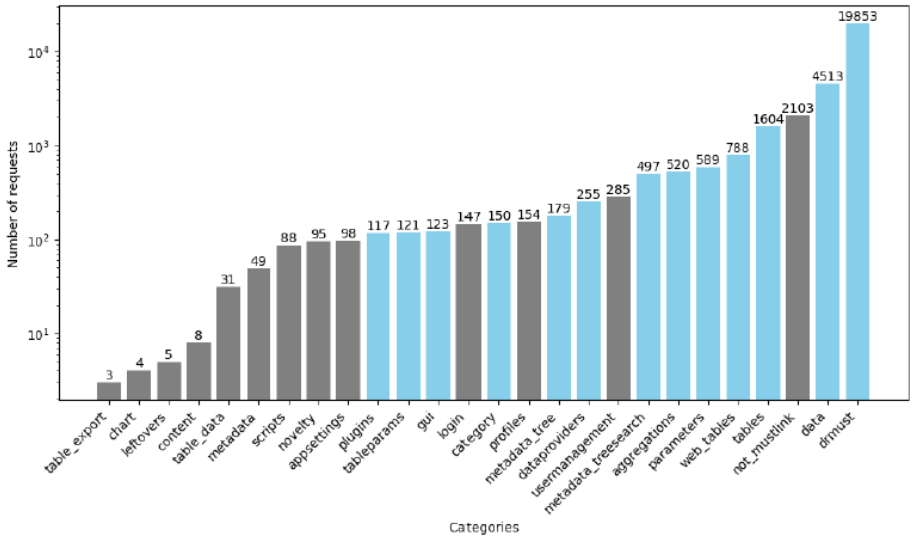

Figure 4. Number of requests for each API endpoint. Blue bars were included in the study. Gray bars were deemed as of-the-scope as they refer to deprecated or superfluous functionality. Note the logarithmic scale.

In MUST, data importation occurs on a scheduled basis, independent of the REST API endpoints available to application clients. We can replicate the operational importation patterns in a simulated environment by using the same cadence observed in practice, as detailed in the schedule configuration files. These files indicate that importation is triggered every 10 minutes, although mission experts report that new data is typically available only every 100 minutes on average.

By combining endpoint requests with the scheduler configuration, we can define the typical operational usage of MUST. For the period from August 12th to August 18th, 2024, we analysed logged requests categorized by the system's components and examined the importation process using a day of data extrapolated to represent a week. This approach provides a comprehensive overview of the system's activity during the specified timeframe.

3.2 Step 2: Setup, Simulate and Measure

We configured MUST on a dedicated server that is kept as clean as possible from background tasks, to ensure accurate energy measurements. With MUST running, we issue synthetic requests following the data curated from the API access logs and measure the energy consumption using a tool called Energibridge [8] - a cross-platform energy measurement utility, to perform measurements. Similarly, we perform all the imports in the typical use case, and measure in the same way.

To ensure accuracy, each request must generate a representative workload. For instance, requesting parameter 'A' from November 1st, 00:00 to November 2nd, 23:59 requires data for those days, resulting in 172,740 data points at a frequency of 1 Hertz. Synthetic requests are carefully reproduced, considering all arguments and intervals. We ensure databases contain the necessary data for all requested parameters.

An 'orchestrator' script runs each category of requests and imports, measuring energy consumption. Each experiment is conducted five times, with results averaged after a 30-second 'warm-up' to stabilize the system. The order of experiments is randomized, and the setup is recreated between runs to avoid biases.

To measure energy consumption during data importation, we assess both empty imports and 14 batches of data over 100 minutes. This results in 126 empty runs and 14 data runs per day. We extrapolate to a week by multiplying the results by 7, as data volume remains consistent.

3.3 Step 3: Discuss Results and Formulate New Experiments

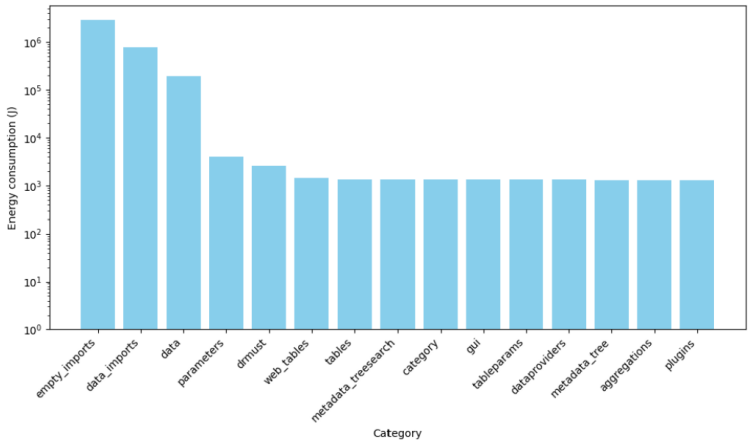

Figure 5. Average Energy Consumption of the different tested components of MUST. Note the logarithmic scale.

Figure 5 shows the average energy consumption for the different MUST components. It provides immediate graphical understanding of the energy consumption hotspots. To make this information more understandable, we created a differentiation between empty and non-empty imports.

According to our measurements, importation is responsible for most of the energy consumption. Moreover, the amount of energy spent in empty imports exceeds the required amount of energy needed to actually import telemetry in MUST.

As expected, parameter data requests have a significant share of the energy usage of the application while the DrMUST endpoint presents as an interesting hotspot to be considered (note the logarithmic scale).

Compared to imports and data requests, all other categories represent an insignificant amount of work: summed together, all other work represents only around 0.5% percent of the total energy consumption. In order to get deeper understanding of these findings, we presented them to MUST engineers.

DrMUST

MUST engineers explained that each user using the web application sends a request to the endpoint every minute. This request is made to the REST API to check for updates on the status of active DrMUST anomaly investigations. However, in our use case, there are no ongoing investigations. The engineers confirmed that DrMUST usage is generally low, and this polling mechanism often yields no results.

As a result, while the energy consumption from these DrMUST requests is relatively low, it is largely unnecessary. Each individual DrMUST request consumes only about 0.13 joules, compared to 42.44 joules for a typical data request. Implementing a WebSocket approach could potentially reduce this energy consumption significantly.

Empty Importation

Figure 6 illustrates the average energy consumption for nine empty imports compared to the average energy used for importing each of the thirteen 100-minute data chunks. These thirteen chunks collectively represent one day of data, with the importer running nine times without data for each chunk. The energy consumption for importing the different data chunks is relatively consistent. While a single empty import uses less energy than a data import, the total energy consumed by nine empty imports exceeds that of a single data import.

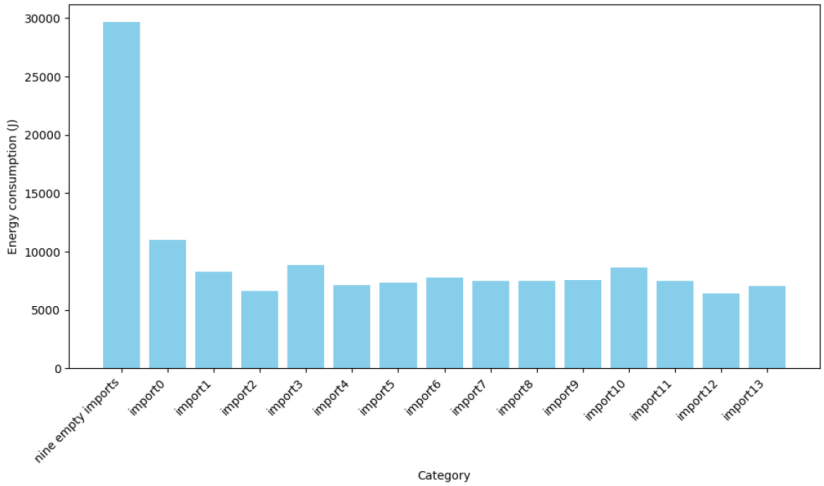

Figure 6. The average energy consumption of the importing the 14 100-minute data chunks, as well as the average energy consumption of the nine empty imports.

It's not surprising that this is a hotspot, as MUST processes large amounts of data, which requires significant time and effort. However, the energy wasted by running the importer without data is substantial. In each 100-minute cycle, the energy used for importing data is less than that used for running the importer without any data.

MUST engineers explained that when the importer starts, it loads various metadata assets, establishes database connections, and initiates a transfer queue, among other tasks. When asked if it could quickly check for new telemetry before fully engaging the importer, the answer revealed that as noted in section 2.1, MUST is a complex system that imports data from various sources, and some requests depend on metadata and configurations that must be loaded first. However, for our use case, which aligns with most current missions, this work is unnecessary. Implementing a quick check for new telemetry before starting energy-intensive tasks would be a sensible approach. While this change may not be straightforward due to the importer base class being used by all data sources, it is in principle justified.

Data Requests

MUST's main use case is to get data from the database and plot it. Because of this, data requests represent the vast majority of work that is triggered by users. This is true both in terms of the number of requests and in terms of the amount of work per request. It makes sense to dive deeper into these data requests, to investigate further what factors play a role in the energy consumption of these data requests and what we can do about that. Based on expert opinion, we identified two key questions:

1. What is the influence of the quantity of data on the energy consumption of the data requests?
2. What is the influence of the various arguments of the data requests on energy consumption?

For question 1., we vary the amount of data that a data request has to process by varying the duration of the requests. At a set frequency, a request for a two-week period of data will contain twice as much data as one for a one-week period of data.

For question 2., we identified three interesting arguments of the data requests API:

- **VRA** stands for Viewport Resolution Aggregation [15]. The underlying idea is simple: if a user requests data to create a graph, and the graph is only 1000 pixels wide, it is not necessary to send hundreds of thousands of datapoints over the network. Instead, given a viewport that is N pixels wide, the algorithm divides all datapoints into N chunks, and returns the minimum and maximum value in each chunk. This reduces the amount of datapoints from (possibly) hundreds of thousands to just 2N. Even a full screen image on a 4K display this is only ±16000. However, this approach comes with (some) overhead of retrieving all the data and finding the minimum and maximum values. MUST can be configured to engage the VRA algorithm once a threshold of a certain number of datapoints is reached. By default, MUST engages VRA when more the response would otherwise contain more than 15000 datapoints.
- **Streaming** This is a boolean argument. Without streaming, the whole result set from the database is transformed into a java object, then into a json object, which is then sent over the network. With streaming,

the database result set is streamed directly into a JSON buffer, which is then sent over the network. This saves memory and a conversion step, which can, for large datasets, have a significant impact.

- **Compact** MUST sends datapoints as a list of json objects of the form data: [{"datetime": <datetime>, "value": <value>}, {...}, ...] when compact is false. When compact is true, the format changes to data: [{"d": <datetime>, "v": <value>}, {...}, ...]. When sending a large number of datapoints, this small optimization can have a major impact. Due to a bug in MUST, in the current implementation, compact only works when streaming is enabled.

3.4 Step 4: Investigate Hotspots and Surprising Results

Starting from an empty database, we create a single test parameter with datapoints in a period of two months, with a frequency of one datapoint every second, with the VRA disabled. Then we perform requests for various periods. We perform requests with periods ranging from 1 to 1000 hours, in steps of 50 hours each. Each experiment is performed 5 times, with a 30-second warm-up period, in a random order, and the results are averaged.

As we are performing the same request over and over again, the system may be able to cache the results, which would invalidate the measurements. To counteract this, we slide the period of a week for which we request data by one second every time we do a request. In this way, every request is different, and the system cannot cache it.

We perform each experiment 5 times, with a 30-second warm-up period each time, and average the results.

VRA

To assess the impact of VRA, we replicate the previous experiment with VRA enabled, using a chunk count - the width of the viewport in pixels - of 1,000 pixels for the algorithm. As noted in section 3.3, VRA activates when a request exceeds 15,000 samples. At a rate of 1 Hz, this threshold is reached in about 4.1 hours. To capture this point more accurately, we conduct requests with periods ranging from 1 to 100 hours in 1-hour increments, testing three settings: VRA enabled at the default threshold, VRA disabled, and VRA always enabled. Each request is performed 1,000 times, and each experiment is repeated five times. We slide the request period by 1 second to avoid caching and include a 30-second warm-up before each experiment.

Streaming

To evaluate the impact of the streaming parameter, we repeat the previous experiment, with request periods ranging from 1-100 hours in steps of 1 hour again, with VRA disabled, but this time with streaming enabled.

Compact

Due to a bug in MUST, the compact parameter only activates when streaming is also on. To measure its impact, we repeat the same experiment, this time with both streaming and compact enabled, and with VRA still disabled.

Findings

Figure 7 reveals that as request periods increase, energy consumption also rises. However, for these long request periods, activating VRA decreases the rate at which energy consumption increases.

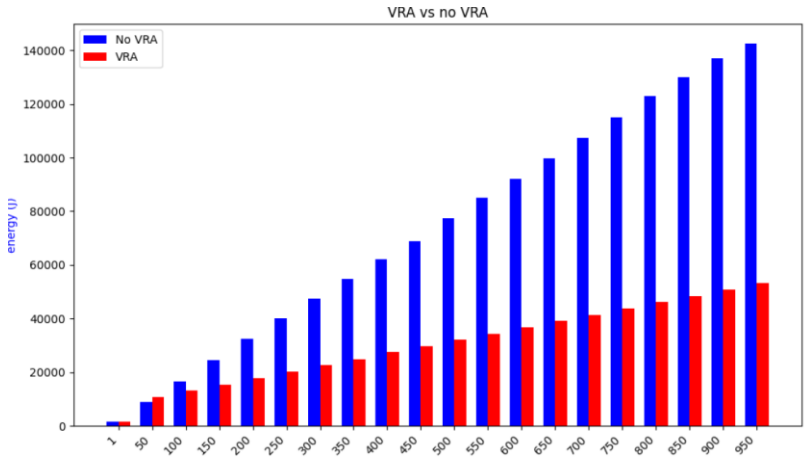

Figure 7. Shows the energy consumption of a request for a parameter with a 1 second frequency, ranging from a 1 hour request period to a 950 hour period in 50 hour steps. For the red bars, VRA was enabled and for the blue bars, it was not.

Figure 8 shows the result of the 'zoomed-in' VRA experiments, with request periods ranging from one to one hundred hours in steps of 1 hour. The plot shows the results of when VRA is always enabled, VRA is enabled from the 15000 hour threshold (which occurs between the four and five hour mark), and with VRA disabled. We see a clear jump, right at this threshold, in the energy consumption when VRA is turned on. Furthermore, before the 65 hour mark (which corresponds to 234000 samples, much more than the default 15000 sample threshold), enabling VRA actually consumes more energy. After the 65 hour mark, these lines cross and VRA improves energy efficiency.

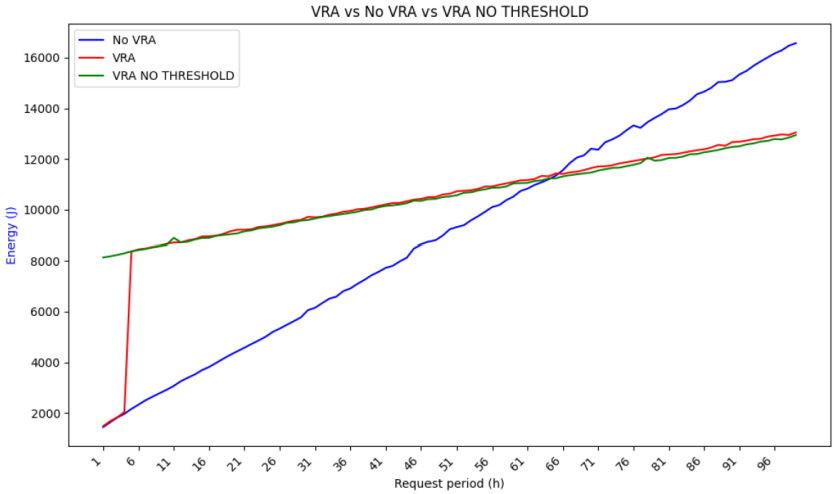

Figure 8. The energy consumption of a request for a parameter with a 1-second frequency, ranging from a 1 hour request period to a 100 hour period in 1 hour steps. The red line has VRA enabled as normally configured, from the threshold of 15000 samples. At 1hz, this threshold is reached between a 4 hour request period (14400 samples) and a 5 hour request period (18000 samples). The green line has VRA enabled always, and the blue line has VRA disabled.

Figure 9 shows the results of the investigations into the streaming and compact parameters. Again, these are requests with a period ranging from 1 hour to 100 hours, in 1 hour steps. The plot shows that streaming is more efficient than not streaming, but that the compact parameter barely makes an impact when enabled in conjunction with streaming. We were unable to test compact separately from compact, due to a bug in MUST.

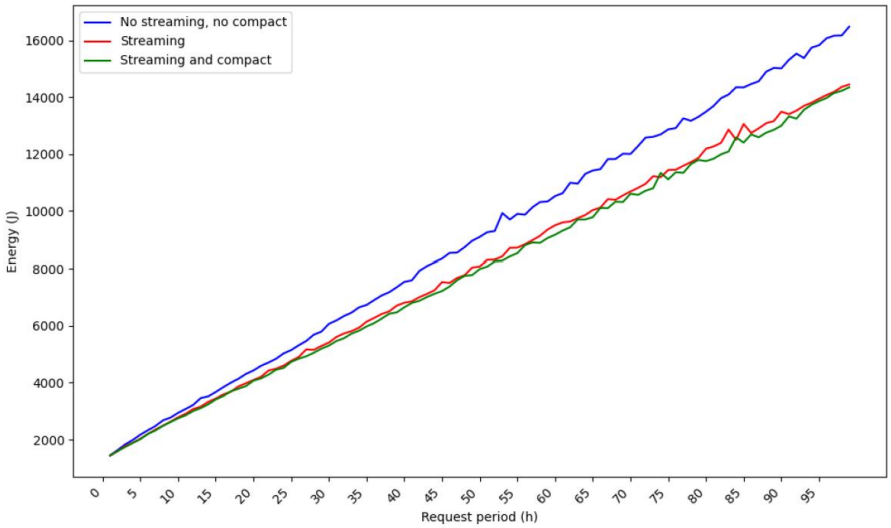

Figure 9. The energy consumption of a request for a parameter with a 1 second frequency, ranging from a 1-hour request period to a 100-hour period in 1-hour steps. For the blue line, requests were done with streaming and compact disabled. With the red line streaming is enabled, and with the green line, both streaming and compact are enabled.

3.5 Step 5: Report and Engage

The results from Step 4 were shared with various stakeholders, including MUST developers, Solenix engineers and managers, the MUST technical officer, and other ESOC staff. From these discussions, we reached the following conclusions:

- **Energy Waste**: Both the importer and DrMUST waste energy. While a comprehensive fix for all importer types will require significant effort, a straightforward solution for file-based importers should be implemented promptly.
- **DrMUST Improvements**: The fix for DrMUST is also relatively simple. We should prioritize addressing the low-hanging fruit identified through the SMURF methodology. Although MUST is a small component of the ground-segment landscape, applying SMURF can effectively reduce energy waste, especially when combined with higher-usage applications.
- **Complex Issues**: More challenging problems, such as optimizing VRA or all importers in MUST, involve complex trade-offs regarding priority, required effort, and available budget. However, we are not yet at a stage where these issues are critical, as there are still many straightforward fixes to address.
- **Policy and Procurement**: To tackle more complex issues, we need to approach them from a policy and procurement perspective. Creating the right incentives for sustainable software is challenging, and while some options have been proposed, further exploration is necessary.

Overall, our discussions highlighted two key points: the results strongly motivate us to address the identified issues and raise awareness of sustainable software practices. This increased awareness can help normalize sustainability discussions in technical meetings, prioritize budget allocation for sustainability fixes, and foster a culture of care for these issues.

4. Discussion

By using server logs to establish a "typical use case", SMURF can accurately compare the energy consumption of different components within a single system version and evaluate systems with multiple core functionalities. This aspect makes SMURF success-oriented: As hotspots result from the analysis of real application usage patterns, then addressing these result in substantial impact.

The involvement of the application engineers enabled the full-characterization of the hotspots identified. If these are now followed accordingly, the fixes can be integrated into the application. These constitute effective gains to be multiplied by the multiple existing deployments across the agency. Thanks to the involvement of MUST engineers and other stakeholders, it was possible to formulate additional experiments that enhance understanding and inform design improvements for the application's energy profile.

Additionally, SMURF serves as an educational tool, raising awareness about energy consumption and sustainability in software design. In our case study, stakeholders - including users, developers, and product owners - expressed that they had not previously considered the sustainability of software systems, but this study sparked their interest. Their familiarity with the system transforms the concept of sustainable software engineering from an abstract idea into a more tangible concern.

These insights can catalyse discussions about sustainability in technical meetings, contributing to a cultural shift within organizations that values energy efficiency and empowers individuals to voice sustainability concerns.

Overall, the results of the SMURF methodology are a powerful tool for raising awareness and engaging with sustainability issues. While many attributes of SMURF suggest it can be applied to various systems, our case study focused on a single system, necessitating further verification. Additionally, SMURF has specific requirements that not all systems will meet, which may limit its applicability or require modifications. It is particularly challenging to apply SMURF to non-deterministic systems, where measuring and comparing energy profiles can be difficult due to variability in processing effort, data produced, and resource requirements.

The main challenges in applying the SMURF methodology include the practical difficulties of setting up a test deployment that accurately reflects the production system, the challenges of performing precise energy measurements, and the fact that while results can identify general trends and hotspots, they will always be relative. Yet, this methodology paves the way to make software systems energy-aware by design, and our case study highlights the importance of involving stakeholders in improving energy efficiency, without requiring specialized expertise in energy efficiency and freeing them from the burden of data collection and analysis.

4.1 Future Work

The first priority is to address the low-hanging fruit identified by the SMURF methodology in MUST by implementing simple changes to the DrMUST endpoint and the importer. This requires support from key stakeholders, all of whom have expressed interest in resolving these issues, making it the top priority. Additionally, during step five of the methodology, MUST developers were surprised by the poor performance of the VRA implementation for smaller requests. Since these smaller requests are the most common, this issue warrants further investigation.

To enhance and validate the SMURF methodology, it should be applied to more systems to determine its broader applicability and allow for necessary modifications.

Moreover, conducting longitudinal studies could provide valuable insights into whether organizations act on the recommendations from SMURF and address energy hotspots, particularly those with a positive cost-benefit impact.

5. Conclusions

Using a systematic five-step approach, the SMURF methodology enables the assessment of energy profiles in multifaceted systems. By leveraging server logs and usage data to establish a typical use case, the methodology remains grounded in reality, even as system complexity increases.

The SMURF methodology identified several hotspots and energy-wasting components within the MUST system, leading to actionable recommendations and renewed motivation for their implementation. Specifically, it revealed inefficiencies in the importation process and the DrMUST functionality. Additionally, the methodology provided insights into the energy implications of specific implementation details. Our investigation of MUST's data endpoint confirmed that existing optimization strategies are effective, particularly for large requests. However, it also uncovered a potential flaw in the Viewport Resolution Aggregation algorithm that requires further examination.

Among the three components raised as hotspots (DrMUST, imports, and data requests), the data endpoint is the only one which has undergone significant optimization efforts – possibly because these optimizations resulted from the users perceiving the performance as not acceptable. When practical considerations align with sustainability goals, resources become available to address these issues. To make meaningful progress on sustainability, it is essential to engage with and raise awareness about these concerns, especially in areas where alignment is lacking.

The application of the SMURF methodology served as an effective tool for promoting engagement and awareness of sustainable software principles. In discussions with developers, users, and product owners, common themes emerged: all stakeholders agreed that the most significant issues with simple fixes - the low-hanging fruit - should be prioritized. Many expressed that they had not previously considered sustainable software engineering, but the SMURF process sparked their interest. They raised concerns about establishing the right incentives to prioritize sustainability issues while recognizing that this perspective provided valuable motivation to address existing problems.

6. Acknowledgements

We thank Solenix and all engineers involved in the case study for the availability, support and continued interest in this study. We would also like to thank June Sallou for the great exchanges while discovering the unknowns of the energy utilization of MUST. Finally, a word to Thomas Ormston and ESOC for the opportunity of using MUST as a case-study for this work.

References

[1] Khan, M. U., Abbas, S., Lee, S. U. J., & Abbas, A. (2021). Measuring power consumption in mobile devices for energy sustainable app development: A comparative study and challenges. *Sustainable Computing: Informatics and Systems*, *31*, 100589.

[2] Li, Y., Lin, Y., Wang, Y., Ye, K., & Xu, C. (2022). Serverless computing: state-of-the-art, challenges and opportunities. *IEEE Transactions on Services Computing*, *16*(2), 1522-1539.

[3] Muhuri, P. K., Gupta, P. K., & Mendel, J. M. (2017). User-satisfaction-aware power management in mobile devices based on perceptual computing. *IEEE Transactions on Fuzzy Systems*, *26*(4), 2311-2323.

[4] Freitag, C., Berners-Lee, M., Widdicks, K., Knowles, B., Blair, G. S., & Friday, A. (2021). The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns*, *2*(9).

[5] Pont, A., Robles, A., & Gil, J. A. (2019). e-WASTE: everything an ICT scientist and developer should know. *IEEE Access*, *7*, 169614-169635.

[6] Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., ... & Clause, J. (2016, May). An empirical study of practitioners' perspectives on green software engineering. In Proceedings of the 38th international conference on software engineering (pp. 237-248).

[7] Solenix. *Revolutionizing Space Operations: The Evolution of MUST | Solenix*. https://solenix.ch/blog/2023-11-01/revolutionizing-space-operations-evolution-must. (Visited on 28/02/2025).

[8] Kaaij, O. K. N. (2025). SMURF: a Methodology for Energy Profiling Software Systems, TU Delft.

[9] Luís Cruz. *Green Software Engineering Done Right: a Scientific Guide to Set Up Energy Efficiency Experiments*. http://luiscruz.github.io/2021/10/10/scientific-guide.html. Blog post. 2021. DOI: 10.6084/m9.figshare.22067846.v1.

[10] Javier Mancebo, Félix García, and Coral Calero. "A Process for Analysing the Energy Efficiency of Software". In: *Information and Software Technology* 134 (June 2021), p. 106560. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2021.106560. (Visited on 12/16/2024).

[11] June Sallou, Luís Cruz, and Thomas Durieux. *EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement*. Dec. 2023. DOI: 10.48550/arXiv.2312.13897. arXiv: 2312.13897 [cs]. (Visited on 12/18/2024)

[12] Santos, R. (2014). How the use of "Big Data" clusters improves off-line data analysis and operations. In *SpaceOps 2014 Conference* (p. 1735).

[13] Martinez, J. (2012). Drmust-a data mining approach for anomaly investigation. In *SpaceOps 2012* (p. 1275109).

[14] Martinez, J. (2012). New telemetry monitoring paradigm with novelty detection. In SpaceOps 2012 (p. 1275123).

[15] Henrique Oliveira et al. "Enabling Visualization of Large Telemetry Datasets". In: *12th International Conference on Space Operations (SpaceOps 2012), Stockholm, Sweden*. 2012, pp. 11–15. (Visited on 01/28/2025).